Programming Assignment 2

Name: Sarvesh Vikas Gharat
Roll Number: 22D1593

October 12, 2022

I. Provided Files

In this task, we need to find an optimal policy for batter ”A” who’s capable of hitting boundaries as opposed to batter ”B” over whom we have no control but can only take 0 or 1 with some probability which is a function of q.
At any time, both batters have some probability of getting out (for all actions of A), after which the game ends.

To perform this task, we have 3 files:

A. Encoder

In this .py file, we take all the states (formatted in bbrr) as an input along with the outcome probabilities for every actions of Player A and ”q” which is degree of weakness of batter ”B” to create a MDP file in desired format.
The states present in the input file doesn’t includes win and loose states, hence we append those 2 states to the state list. Also, to reduce numerical calculations (for transition probabilities), in this algorithm we consider the total number of states to be $2\times$ number of states which we received in input + Win + Loose. The first set of states considers ”A” being on strike, hence it can be considered as ”Abbrr”. Similarly the other set of states excluding win and loose considers ”B” being on strike i.e ”Bbbrr”. Therefore, in case of 15 balls and 10 runs, the total number of steps will be ”150 + 150 + 2 = 302”.
Further, as we already have probabilities of outcomes for corresponding actions modelling the MDP becomes an easier task, as the given probability happens to be the transition probability eg: Consider A tries to take 1 (action 1) and gets 1 with probability ”x” from state ”1510” than we have current state as ”A1510”, action as 1, next state as ”B1409” and transition probability as ”x”.
Further as ”B” can only take 0, 1 or get out with some probability as a function of ”q”, we temporarily create a new action which ”B” will always take, the next state for which will depend on corresponding probabilities. Hence, although we have some extra states of ”B”, it won’t create any computational complexity while determining the optimal policy as ”B” can only take 1 action.
The reward from every state to every other states (besides win) is considered as 0, similarly on winning, we get the reward of 1.

B. Planner

It’s the same file which we created for task 1. This files consists of three algorithms i.e Value Iteration, Howard Policy Iteration and Linear Programming to get optimal policy. To implement Linear Programming, we make use of Pulp. The precision as required for VI is kept to be 10^{-8} and the default algorithm which helps in getting the optimal policy for this task is ”HPI”.

C. Decoder

As we had appended some extra states, it’s important to remove those before printing out the value function and the corresponding action for ”A”. Hence, in the decoder.py, we remove all the temporary states which were created before running the planner and print the optimal policy for ”A” along with the value function at every state.

II. Graphs and Observations

In this section, we discuss 3 cases:

A. Varying ”q”

Here, we fix out state to be 1530 i.e 30 runs required in 15 balls, and observe the variation in winning probability on varying ”q”

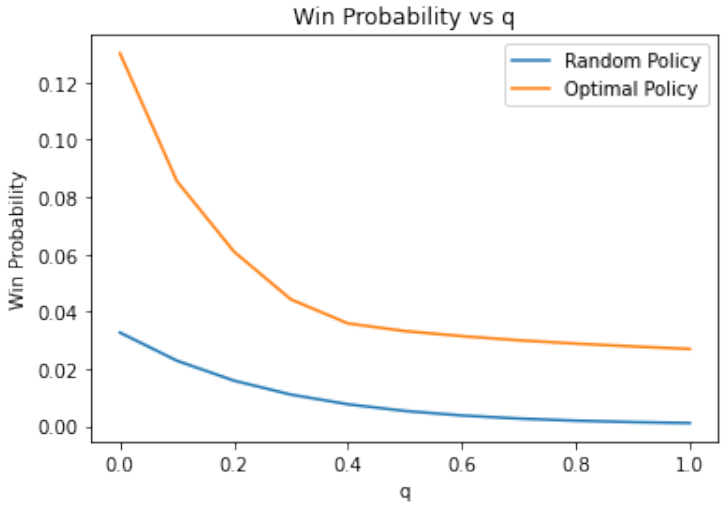

Figure 1: Win Probability vs degree of weakness (q)

As seen in Figure 1, for ”q = 0”, we get maximum winning probability as the chances of batter ”B” getting out is 0. As we move ahead, the winning probability goes on decreasing due to higher probability of Batsman ”B” getting weaker and weaker and hence loosing his wicket. Hence, in case of q = 1, in any situation if batting team has to win, the Batter ”A” always needs to score even number of runs provided it’s not the last ball of the over (in this case odd number of runs are needed to get the strike back). We also see that, the optimal policy generated by our algorithm is relatively way better than the random policy given.

B. Varying ”runs required”

In this case, the number of balls are fixed to ”10” along with q as ”0.25”. Here, we observe the variation in winning probability on varying the number of runs required.

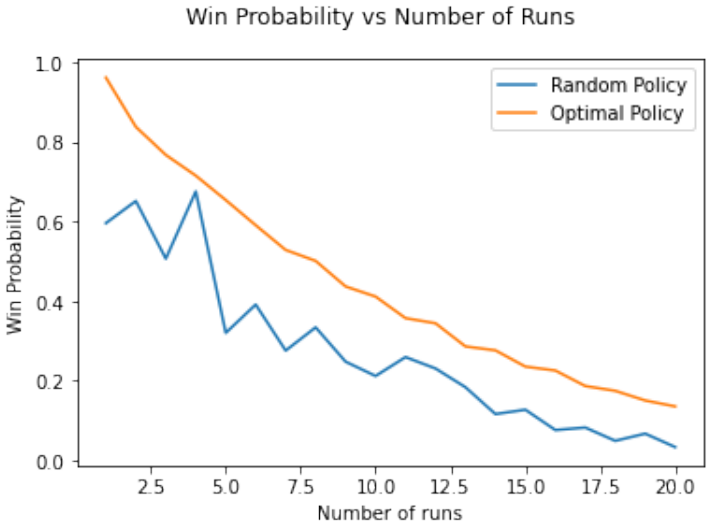

Figure 2: Win Probability vs Runs required

As seen in Figure 2, as the required number of runs increase, the probability of winning goes on decreasing which is obvious due to constant number of balls available. We also see that the optimal policy as generated by our algorithm performs way better as compared to random policy which was generated.

C. Varying ”balls left”

In this case, the number of balls are fixed as ”10” along with q as ”0.25”. Here, we observe the variation in winning probability on varying the number of balls left.

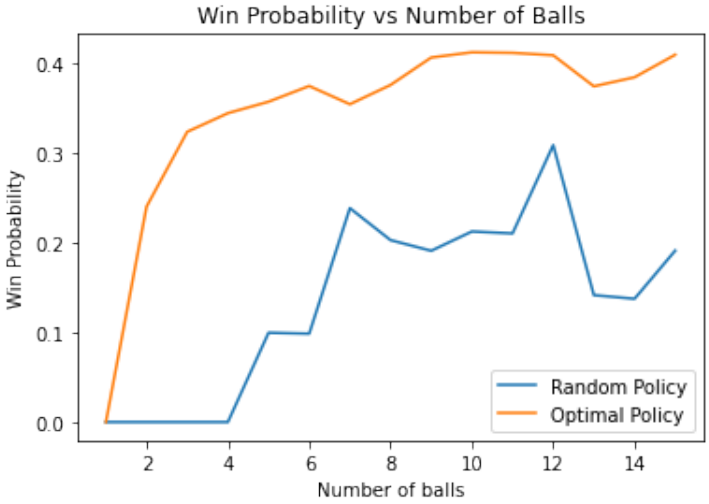

Figure 3: Win Probability vs Balls left

As seen in Figure 3, on increasing the number of balls, the probability of winning increases. We also see that at some point, the graph shows a sudden dip in the win probability, this is due to possibility of changing the strike after end of the particular over. We also see that on having ”number of balls = 1” the probability of winning the match becomes 0. This is due to the maximum outcome or the maximum number of scorable runs on any fair delivery being 6. Hence,

it is impossible to score 10 with 1 ball remaining unless the delivered ball is unfair which is not the case covered in this task. We also observe that the optimal policy as generated by our algorithm performs better than the random policy which was generated. This can be seen in all the 3 cases. Hence, we can say the optimal policy generated by our algorithm is better than that of randomly generated policy.

As in our case, we consider total number of states to be 2 times the given states + 2, here we need to convert the states of random policy in similar format. Hence to do that we write a separate python code which appends ”A” to all the given states and than creates the same number of states for ”B” with one constant action. As it’s an episodic task, the terminal states i.e Win and Loose have no further actions.